



Runtime randomization and perturbation for virtual machines.

JAVIER CABRERA ARTEAGA

Licentiate Thesis in [Research Subject - as it is in your ISP]
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden [2022]

TRITA-ICT XXXX:XX
ISBN XXX-XX-XXXX-XXX-X

KTH School of Information and
Communication Technology
SE-164 40 Kista
SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägg-
es till offentlig granskning för avläggande av licentiatexamen i [ämne/subject]
[veckodag/weekday] den [dag/day] [månad/month] [år/2022] klockan [tid/time] i
[sal/hall], Electrum, Kungl Tekniska högskolan, Kistagången 16, Kista.

© Javier Cabrera Arteaga, [month] [2022]

Tryck: Universitetsservice US AB

Abstract

Write your abstract here...

Keywords: Keyword1, keyword2, ...

Sammanfattning

Write your Swedish summary (popular description) here...

Keywords: Keyword1, keyword2, ...

Acknowledgements

Write your professional acknowledgements here...

Acknowledgements are used to thank all persons who have helped in carrying out the research and to the research organizations/institutions and/or companies for funding the research.

Name Surname,
Place, Date

Contents

Contents	vi
List of Figures	vii
List of Tables	viii
List of Acronyms	ix
1 Introduction	1
1.1 Motivation	1
1.1.1 Why variants ?	1
1.1.2 Research questions	1
1.2 Contributions	1
2 Background and State of the art	3
3 Methodology	5
3.1 RQ1. To what extent can we artificially generate program variants for WebAssembly?	6
3.2 RQ2. To what extent are the generated variants dynamically different?	9
3.3 RQ3. To what extent do the artificial variants exhibit different exe- cution times on Edge-Cloud platforms?	12
4 Results	15
4.1 RQ1. To what extent can we artificially generate program variants for WebAssembly?	15
4.2 RQ2. To what extent are the generated variants dynamically different?	18
4.3 RQ3. To what extent do the artificial variants exhibit different exe- cution times on Edge-Cloud platforms?	21
Bibliography	25

List of Figures

3.1	The program variants generation for RQ1.	8
3.2	Dynamic analysis for RQ2.	10
3.3	Multivariant binary creation and workflow for RQ3 answering.	12
4.1	Pairwise comparison of programs' population traces in logarithmic scale. Each vertical group of blue dots represents a programs' population. Each dot represents a comparison between two program execution traces according to Metric 5.	19
4.2	Execution time distributions for Hilber_curve program and its variants. Baseline execution time mean is highlighted with the magenta horizontal line.	20
4.3	Execution time distributions. Each subplot represents the quantile-quantile plot of the two distributions, original and multivariant binary.	23

List of Tables

3.1	Corpora description. The table is composed by the name of the corpus, the number of modules, the number of functions, the lines of code range and the location of the corpus.	7
4.1	General program's populations statistics. The table is composed by the name of the corpus, the number of functions, the number of succesfully diversified functions, the cumulative number of generated variants and the cumulative number of unique variants.	16

List of Acronyms

Wasm
DTW

WebAssembly
Dynamic Time Warping

Chapter 1

Introduction

Write a short introduction here...

1.1 Motivation

1.1.1 Why variants ?

1.1.2 Research questions

RQ1. To what extent can we artificially generate program variants for WebAssembly?

T

RQ2. To what extent are the generated variants dynamically different?

RQ3. To what extent do the artificial variants exhibit different execution times on Edge-Cloud platforms?

1.2 Contributions

Chapter 2

Background and State of the art

Chapter 3

Methodology

In this chapter, we present our methodology to answer the research questions enunciated in Subsection 1.1.2. We investigate three research questions. In the first question, we artificially generate WebAssembly program variants and quantitatively compare the static differences between variants. Our second research question focuses on comparing their behavior during their execution. The final research question evaluates the feasibility of using the program variants in security-sensitive environments such as Edge-Cloud computing proposing a multivariant execution approach.

The main objective of this thesis is to study the feasibility of automatically creating program variants out of preexisting program sources. To achieve this objective, we use the empirical method [1], proposing a solution and evaluating it through quantitative analyzes in case studies. We follow an iterative and incremental approach on the selection of programs for our corpora. To build our corpora, we find a representative and diverse set of programs to generalize, even when it is unrealistic following an empirical approach, as much as possible our results. We first enunciate the corpora we share along this work to answer our research questions. Then, we establish the metrics for each research question, set the configuration for the experiments, and describe the protocol.

Corpora

Our experiments assess the impact of artificially created diversity in terms of number of created variants and their static and dynamic differences. The first step is to build a suitable corpus of programs' seeds to generate the variants. Then, we answer all our research questions with three corpora of diverse and representative programs for our experiments. We build our three corpora in an escalating strategy. The first corpus is diverse and contains simple programs in terms of code size, making them easy to manually analyze. The latter two corpora contain more extensive real-world programs, including one project meant for security-sensitive

applications. All corpora are considered to come along the LLVM pipeline. We base this decision on the previous experimental work of Hilbig et al. [2]. This work shows that approximately 65% of all WebAssembly programs come out of C/C++ source code through the LLVM pipeline, and more than 75% if the Rust language is included. In the following, we describe the filtering and description of each corpus.

1. **Rosetta** : We take programs from the Rosetta Code project¹. This website hosts a curated set of solutions for specific programming tasks in various programming languages. It contains many tasks, from simple ones, such as adding two numbers, to complex algorithms like a compiler lexer. We first collect all C programs from the Rosetta Code, representing 989 programs as of 01/26/2020. We then apply several filters: the programs should successfully compile and, they should not require user inputs to automatically execute them, the programs should terminate and should not result in non-deterministic results.

The result of the filtering is a corpus of 303 C programs. All programs include a single function in terms of source code. These programs range from 7 to 150 lines of code and solve a variety of problems, from the *Babbage* problem to *Convex Hull* calculation.

2. **Libsodium**: This project is encryption, decryption, signature, and password hashing library ported to WebAssembly in 102 separated modules. The modules have between 8 and 2703 lines of code per function. This project is selected based on two main criteria: first, its importance for security-related applications, and second, its suitability to collect the modules in LLVM intermediate representation.
3. **QRCode**: This project is a QRCode and MicroQRCode generator written in Rust. This project contains 2 modules having between 4 and 725 lines of code per function. As Libsodium, we select this project due to its suitability for collecting the modules in their LLVM representation. Besides, this project increases the complexity of the previously selected projects due to its integration with the generation of images.

In Table 3.1 we listed the corpus name, the number of modules, the total number of functions, the range of lines of code, and the original location of the corpus.

3.1 RQ1. To what extent can we artificially generate program variants for WebAssembly?

This research question investigates whether we can artificially generate program variants for WebAssembly. We use CROW to generate variants from an original

¹http://www.rosettacode.org/wiki/Rosetta_Code

Corpus	No. modules	No. functions	LOC range	Location
Rosetta	-	303	7 - 150	http://rosettacode.org/wiki/Rosetta_Code
Libsodium	102	869	8 - 2703	https://github.com/jedisct1/libsodium
QrCode	2	1849	4 - 725	https://github.com/kennytm/qr-code-rust
Total		3021		

Table 3.1: Corpora description. The table is composed by the name of the corpus, the number of modules, the number of functions, the lines of code range and the location of the corpus.

program, written in C/C++ in the case of Rosetta corpus and LLVM bitcode modules in the case of Libsodium and QrCode. In Figure 3.1 we illustrate the workflow to generate WebAssembly program variants. We pass each function of the corpora to CROW as a program to diversify. To answer RQ1, we study the outcome of this pipeline, the generated WebAssembly variants.

Metrics

To assess our approach’s ability to generate WebAssembly binaries that are statically different, we compute the number of variants and the number of unique variants for each original function of each corpus. On top, we define the aggregation of these former two values to quantitatively evaluate RQ1 at the corpus level.

Definition 1 *Program’s population $M(P)$: Given a program P and its generated variants v_i , the program’s population is defined as.*

$$M(P) = \{v_i \text{ where } v_i \text{ is a variant of } P\}$$

Notice that, the program’s population includes the original program P .

Definition 2 *Program’s unique population $U(P)$: Given a program P and its program’s population $M(P)$, the program’s unique population is defined as.*

$$U(P) = \{v \in M(P)\}$$

such that $\forall v_i, v_j \in U(P)$, $md5sum(v_i) \neq md5sum(v_j)$. $md5sum(v)$ is the md5 hash calculated over the byte stream of the program file v . Notice that, the original program P is included in $U(P)$.

Metric 1 *Program’s population size $S(P)$: Given a program P and its program’s population $M(P)$ according to Definition 1, the program’s population size is defined*

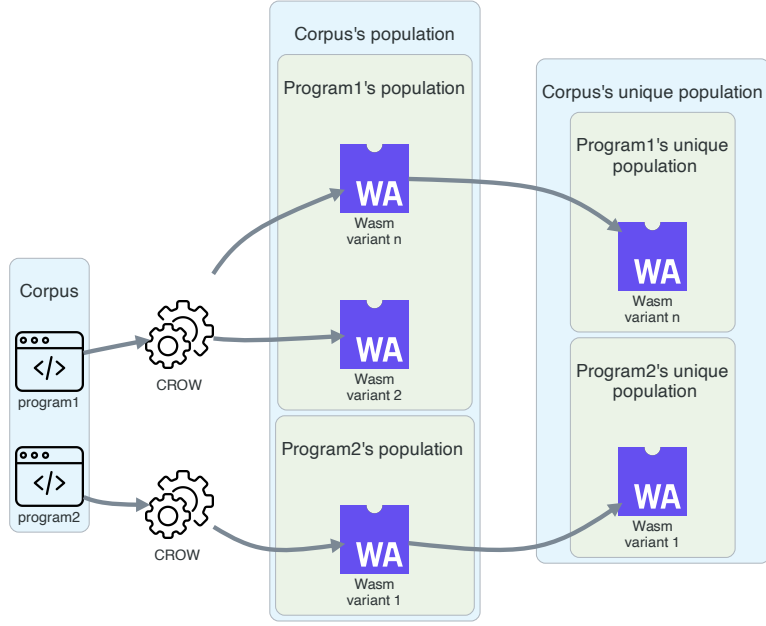


Figure 3.1: The program variants generation for RQ1.

as.

$$S(P) = |M(P)|$$

Metric 2 *Program's unique population size $US(P)$* : Given a program P and its program's unique population $U(P)$ according to Definition 2, the program's unique population size is defined as.

$$US(P) = |U(P)|$$

Metric 3 *Corpus population size $CS(C)$* : Given a program's corpus C , the corpus population size is defined as the sum of all program's population sizes over the corpus C .

$$CS(C) = \sum S(P) \forall P \in C$$

Metric 4 *Corpus unique population size* $UCS(C)$:

Given a program’s corpus C , the corpus unique population size is defined as the sum of all program’s unique population sizes over the corpus C

$$UCS(C) = \sum US(P) \forall P \in C$$

Protocol

To generate program variants, we synthesize program variants with an enumerative strategy, checking each synthesis for equivalence modulo input [3] against the original program. An enumerative synthesis is a brute-force approach to generate program variants. With a maximum number of instructions, it constructs and checks all possible programs up to that limit. For a simplified instance, with a maximum code size of 2 instructions in a programming language with L possible constructions, an enumerative synthesizer builds all $L \times L$ combinations finding program variants. For obvious reasons, this space is nearly impossible to explore in a reasonable time as soon as the limit of instructions increases. Therefore, we use two parameters to control the size of the search space and hence the time required to traverse it. On the one hand, one can limit the size of the variants. On the other hand, one can limit the set of instructions used for the synthesis. In our experiments for RQ1, we use all the 60 supported instructions in our synthesizer.

The former parameter allows us to find a trade-off between the number of variants that are synthesized and the time taken to produce them. For the current evaluation, given the size of the corpus and the properties of its programs, we set the exploration time to 1 hour maximum per function for Rosetta . In the cases of Libsodium and QRCode, we set the timeout to 5 minutes per function. The decision behind the usage of lower timeout for Libsodium and QRCode is motivated by the properties listed in Table 3.1. The latter two corpora are remarkably larger regarding the number of instructions and functions count.

We pass each of the $303 + 869 + 1849$ functions in the corpora to CROW, as Figure 3.1 illustrates, to synthesize program variants. We calculate the *Corpus population size*(Metric 3) and *Corpus unique population size*(Metric 4) for each corpus and conclude by answering RQ1.

3.2 RQ2. To what extent are the generated variants dynamically different?

In this second research question, we investigate to what extent the artificially created variants are dynamically different between them and the original program. To conduct this research question, we could separate our experiments into two fields as Figure 3.2 illustrates: static analysis and dynamic analysis. The static analysis focuses on the appreciated differences among the program variants, as well as between the variants and the original program, and we address it in answering RQ1.

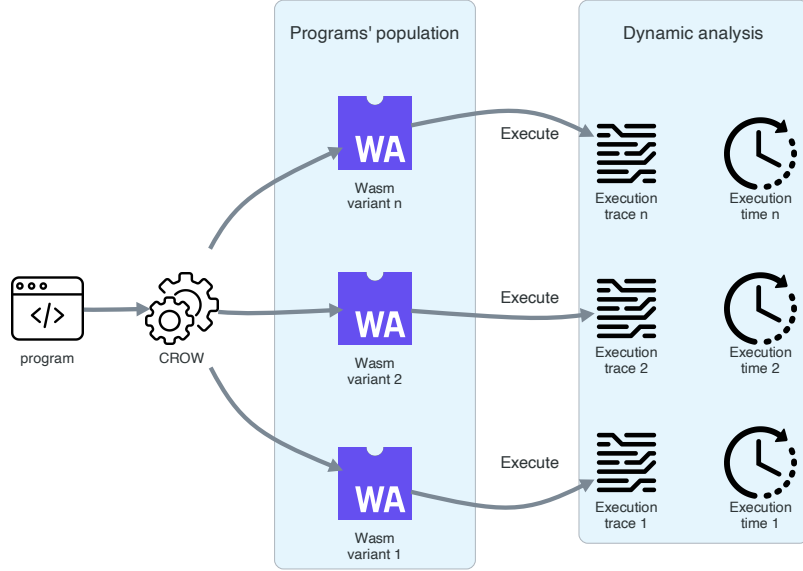


Figure 3.2: Dynamic analysis for RQ2.

With RQ2, we focus on the last category, the dynamic analysis of the generated variants. This decision is supported because dynamic analysis complements RQ1 and, it is essential to provide a full understanding of diversification. We use the original functions from Rosetta corpus described in Section 3 and their variants generated to answer RQ1. We use only Rosetta to answer RQ2 because this corpus is composed of simple programs that can be executed directly without user interaction, *i.e.*, we only need to call the interpreter passing the WebAssembly binary to it.

To dynamically compare programs and their variants, we execute each program on each programs' population to collect their execution traces and execution times. We perform fine-grained comparisons by comparing the traces and execution times for all pairs of programs. Therefore, the defined metrics are formulated to support a pairwise comparison strategy. In the following, we define the metrics used to answer RQ2.

Metrics

We compare the execution traces of two any programs of the same population with a global alignment metric. We propose a global alignment approach using Dynamic Time Warping (DTW). Dynamic Time Warping [4] computes the global alignment between two sequences. It returns a value capturing the cost of this alignment, which is a distance metric. The larger the DTW distance, the more different the two sequences are. In our experiments, we define the traces as the sequence of the stack operations during runtime, *i.e.*, the consecutive list of **push** and **pop** operations performed by the WebAssembly engine during the execution of the program. In the following, we define the *TraceDiff* metric.

Metric 5 *TraceDiff*: Given two programs P and P' from the same program's population, $\text{TraceDiff}(P, P')$, computes the DTW distance between the stack operation traces collected during their execution.

A *TraceDiff* of 0 means that both traces are identical. The higher the value, the more different the traces.

Moreover, we use the execution time distribution of the programs in the population to complement the answer to RQ2. For each program pair in the programs' population, we compare their execution time distributions. We define the execution time as follows:

Metric 6 *Execution time*: Given a WebAssembly program P , the execution time is the time spent to execute the binary.

Protocol

To compare program and variants behavior during runtime, we analyze all the unique program variants generated to answer RQ1 in a pairwise comparison using the value of aligning their execution traces (Metric 5). We use SWAM² to execute each program and variant to collect the stack operation traces. SWAM is a WebAssembly interpreter that provides functionalities to capture the dynamic information of WebAssembly program executions, including the virtual stack operations. We want to remark that we only collect the stack operation traces due to the memory-agnosticism of our approach to generate variants. Our approach does not change the memory-like operations of the original code.

Furthermore, we collect the execution time, Metric 6, for all programs and their variants. We compare the collected execution time distributions between programs using a Mann-Whitney U test [5] in a pairwise strategy.

²<https://github.com/satabin/swam>

3.3 RQ3. To what extent do the artificial variants exhibit different execution times on Edge-Cloud platforms?

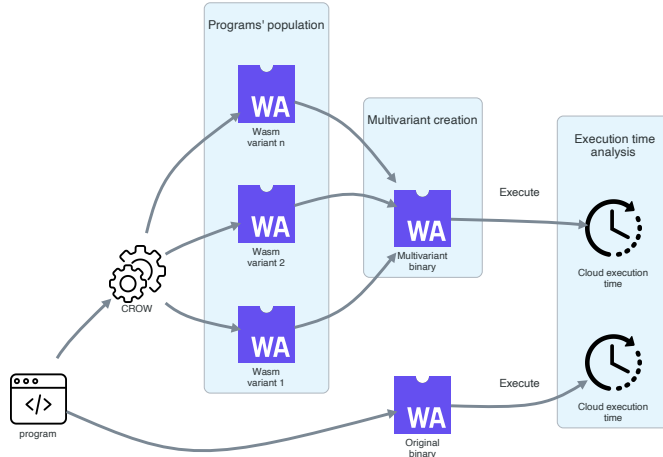


Figure 3.3: Multivariant binary creation and workflow for RQ3 answering.

In the last research question, we study whether the created variants can be used in real-world applications and what properties offer the composition of the variants as multivariant binaries. We build multivariant binaries, and we deploy and execute them at the Edge. The process of *mixing* multiple variants into one multivariant binary is an essential contribution of the thesis that is presented in details in [6]. RQ3 focuses on analyzing the impact of this contribution on execution times. To answer RQ3, we use the variants generated for the programs of Libsodium and QRCode corpora, we take 2 + 5 programs interconnecting the LLVM bitcode modules (mentioned in Table 3.1). We illustrate the protocol to answer RQ3 in Figure 3.3 starting from the creation of the programs' population.

Metrics

We use the execution time of the multivariant binaries to answer RQ3. We use the same metric defined in Metric 6 for the execution time of multivariant binaries.

Protocol

We run the experiments to answer RQ3 on the Edge, executing the multivariant binaries as end-to-end HTTP services. The execution times are measured at the

backend space, *i.e.*, we collect the execution times inside the Edge node and not from the client computer. Therefore, we instrument the binaries to return the execution time as an HTTP header. We do this process for the original program and its multivariant binary. We deploy and execute the original and the multivariant binaries on 64 edge nodes located around the world.

We collect 100k execution times for each binary, both the original and multivariant binaries. We perform a Mann-Whitney U test [5] to compare both execution time distributions. If the P-value is lower than 0.05, the two compared distributions are different.

Conclusions

This chapter presents the methodology we follow to answer our three research questions. We first describe and propose the corpora of programs used in this work. We propose to measure the ability of our approach to generate variants out of 3021 functions of our corpora. Then, we suggest using the generated variants to study to what extent they offer different observable behavior through dynamic analysis. We propose a protocol to study the impact of the composition variants in a multivariant binary deployed at the Edge. Nevertheless, we enumerate and enunciate the properties and metrics that might lead us to answer the impact of automatic diversification for WebAssembly programs. In the next chapter, we present and discuss the results obtained with this methodology.

Chapter 4

Results

In this chapter, we sum up the results of the research of this thesis. We illustrate the key insights and challenges faced in answering each research question. To obtain our results, we followed the methodology formulated in Chapter 3.

4.1 RQ1. To what extent can we artificially generate program variants for WebAssembly?

As we describe in Section 3.1, our first research question aims to answer how to artificially generate WebAssembly program variants. This section is organized as follows. First we present the general results calculating the *Corpus population size* (Metric 3) and *Corpus unique population size* (Metric 4) for each corpus. Second, we discuss the challenges and limitations in program variants generation. Finally, we illustrate the most common code transformations performed by our approach and answer RQ1.

Program’s populations

We summarize the results in Table 4.1. The table illustrates the corpus name, the number of functions to diversify, the number of successfully diversified functions (functions with at least one artificially created variant), the cumulative number of variants (*Corpus population size*) and the cumulative number of unique variants (*Corpus unique population size*).

We produce at least one unique program variant for 239/303 single function programs for Rosetta with one hour for a timeout. For the rest of the programs (64/303), the timeout is reached before CROW can find any valid variant. In the case of Libsodium and QrCode, we produce variants for 85/869 and 32/1849 functions respectively, with 5 minutes per function as timeout. The rest of the functions resulted in timeout before finding function variants or produce no variants. For all programs in all corpora, we achieve 356/3021 successfully diversified functions, rep-

representing a 11.78% of the total. As the four and fifth columns show, the number of artificially created variants and the number of unique variants are larger than the original number of functions by one order of magnitude. In the case of Rosetta , the corpus population size is close to one million of programs.

Corpus	#Functions	# Diversified	# Variants	# Unique Variants
Rosetta	303	239	809900	2678
Libsodium	869	85	4272	3805
QrCode	1849	32	6369	3314
	3021	356	820541	9797

Table 4.1: General program’s populations statistics. The table is composed by the name of the corpus, the number of functions, the number of succesfully diversified functions, the cumulative number of generated variants and the cumulative number of unique variants.

Challenges for automatic diversification

We have observed a remarkable difference between the number of successfully diversified functions versus the number of failed-to-diversify functions (third column of Table 4.1). Our approach successfully diversified 239/303, 85/869 and 32/1849 of the original functions for Rosetta , Libsodium and QrCode respectively. The main reason of this phenomenon is the set timeout for CROW. Setting up the timeout affects the capacity of our approach to generate variants. For our corpora, a low timeout implies a low number of diversified functions.

We have noticed a remarkable difference between the number of diversified functions for each corpus, 809900 programs for Rosetta 4272 for Libsodium and 6369 for QrCode. The corpus population size for Rosetta is two orders of magnitude larger compared to the other two corpora. The reason behind the large number of variants for Rosetta is that, after certain time, our approach starts to combine the code replacements to generate new variants. However, looking at the fifth column, the number of unique variants have the same order of magnitude for all corpora. The variants generated out of the combination of several code replacements are not necessarily unique. Some code replacements can dominate over others, generating the same WebAssembly programs.

A low timeout offers more unique variants compared to the population size despite the low number of diversified functions, like the Libsodium and QrCode cases. This happens because, CROW first generates variants out of single code replacements and then starts to combine them. Thus, more unique variants are generated in the very first moments of the diversification process with CROW.

Apart from the timeout and the combination of variants phenomena, we manually analyze programs, searching for properties attempting to the generation of program variants using CROW. We identify another challenge for diversification. We have observed that our approach searches for a constant replacement for more than 45% of the instructions of each function while constant values cannot be inferred. For instance, constant values cannot be inferred for memory load operations because our tool is oblivious to a memory model.

Properties for large diversification

We manually analyzed the programs to study the critical properties of programs producing a high number of variants. This reveals one key factor that favors many unique variants: the presence of bounded loops. In these cases, we synthesize variants for the loops by replacing them with a constant, if the constant inferring is successful. Every time a loop constant is inferred, the loop body is replaced by a single instruction. This creates a new, statically different program. The number of variants grows exponentially if the function contains nested loops for which we can successfully infer constants.

A second key factor for synthesizing many variants relates to the presence of arithmetic. The synthesis engine used by our approach, effectively replaces arithmetic instructions with equivalent instructions that lead to the same result. For example, we generate unique variants by replacing multiplications with additions or shift left instructions (Listing 4.1). Also, logical comparisons are replaced, inverting the operation and the operands (Listing 4.2). Besides, our implementation can use overflow and underflow of integers to produce variants (Listing 4.3), using the intrinsics of the underlying computation model.

Listing 4.1: Diversification through arithmetic expression replacement.

```
local.get 0    local.get 0
i32.const 2    i32.const 1
i32.mul        i32.shl
```

Listing 4.2: Diversification through inversion of comparison operations.

```
local.get 0    i32.const 11
i32.const 10   local.get 0
i32.gt_s       i32.le_s
```

Listing 4.3: Diversification through overflow of integer operands.

```
i32.const 2    i32.const 2
i32.mul        i32.mul
i32.const      i32.const
                -2147483647
i32.mul
```

Answer to RQ1.

We can provide diversification for 11.78% of the programs in our corpora. Constant inferring, complemented with the high presence of arithmetic operations and bounded loops in the original program increased the number of program variants. Nevertheless, the combination of code replacements in our approach is not a determinant factor to provide a large number unique program variants.

4.2 RQ2. To what extent are the generated variants dynamically different?

Our second research question investigates the differences between program variants at runtime. To answer RQ2, we execute each program/variant generated to answer RQ1 for Rosetta corpus to collect their execution traces and execution times. For each programs' population we compare the stack operation traces (Metric 5) and the execution time distributions (Metric 6) for each program/variant pair.

This section is organized as follows. First, we analyze the programs' populations by comparing the traces for each pair of program/variant with TraceDiff of Metric 5. The pairwise comparison will hint at the results at the population level. We analyze not only the differences of a variant regarding its original program, we also compare the variants against other variants. Second, we do the same pairwise strategy for the execution time distributions Metric 6, performing a Mann-Whitney U test for each pair of program/variant times distribution. Finally, we conclude and answer RQ2.

Stack operation traces.

In Figure 4.1 we plot the distribution of all comparisons (in logarithmic scale) of all pairs of program/variant in each programs' population. All compared programs are statically different. Each vertical group of blue dots represents all the pairwise comparison of the traces (Metric 5) for a program of Rosetta corpus for which we generate variants. Each dot represents a comparison between two programs' traces according to Metric 5. The programs are sorted by their number of variants in descending order. For the sake of illustration, we filter out those programs for which we generate only 2 unique variants.

We have observed that in the majority of the cases, the mean of the comparison values is remarkably large. We analyze the length of the traces, and one reason behind such large values of TraceDiff is that some variants result from constant inferring. For example, if a loop is replaced by a constant, instead of several symbols in the stack operation trace, we observe one. Consequently, the distance between two program traces is significant.



Figure 4.1: Pairwise comparison of programs' population traces in logarithmic scale. Each vertical group of blue dots represents a programs' population. Each dot represents a comparison between two program execution traces according to Metric 5.

In some cases, we have observed variants that are statically different for which TraceDiff value is zero, *i.e.*, they result in the same stack operation trace. We identified two main reasons behind this phenomenon. First, the code transformation that generates the variant targets a non-executed or dead code. Second, some variants have two different instructions that trigger the same stack operations. For example, the code replacements below illustrate the case.

(1) <code>i32.lt_u</code>	<code>i32.lt_s</code>	(3) <code>i32.ne</code>	<code>i32.lt_u</code>
(2) <code>i32.le_s</code>	<code>i32.lt_u</code>	(4) <code>local.get 6</code>	<code>local.get 4</code>

In the four cases, the operators are different (original in gray color and the replacement in green color) leaving the same values for equal operands. The (1) and (2) cases are comparison operations leaving the value 0 or 1 in the stack taking into account the sign of their operands. In the third case, the replacement is less restricted to the original operator, but in both cases, the codes leave the same value in the stack. In the last case, both operands load a value of a local variable in the stack, the index of the local variable is different but the value of the variable that is appended to the trace is the same in both cases.

Execution times.

Even when two programs of the same population offer different execution traces, their execution times can be similar (statistically speaking). In practice, the execution traces of WebAssembly programs are not necessarily accessible, being not the case with the execution time. For example, in our current experimentation we need to use our own instrumentation of the execution engine to collect the stack trace operations while the execution time is naturally accessible in any execution

environment. This mentioned reasoning enforces our comparison of the execution times for the generated variants. For each program’s population, we compare the execution time distributions, Metric 6, of each pair of program/variant. Overall diversified programs, 169 out of 239 (71%) have at least one variant with a different execution time distribution than the original program (P-value < 0.01 in the Mann-Withney test). This result shows that we effectively generate variants that yield significantly different execution times.

By analyzing the data, we observe the following trends. First, if our tool infers control-flows as constants in the original program, the variants execute faster than the original, sometimes by one order of magnitude. On the other hand, if the code is augmented with more instructions, the variants tend to run slower than the original.

In both cases, we generate a variant with a different execution time than the original. Both cases are good from a randomization perspective since this minimizes the certainty a malicious user can have about the program’s behavior. Therefore, a deeper analysis of how this phenomenon can be used to enforce security will be discussed in answering RQ3.

To better illustrate the differences between executions times in the variants, we dissect the execution time distributions for one programs’ population of Rosetta . The plots in Figure 4.2 show the execution time distributions for the **Hilbert_curve** program and their variants. We illustrate time diversification with this program because, we generate unique variants with all types of transformations previously discussed in Section 4.1. In the plots along the X-axis, each vertical set of points represents the distribution of 100000 execution times per program/variant. The Y-axis represents the execution time value in milliseconds. The original program is highlighted in green color: the distribution of 10000 execution times is given on the left-most part of the plot, and its median execution time is represented as a horizontal dashed line. The median execution time is represented as a blue dot for each execution time distribution, and the vertical gray lines represent the entire distribution. The bolder gray line represents the 75% interquartile. The program variants are sorted concerning the median execution time in descending order.

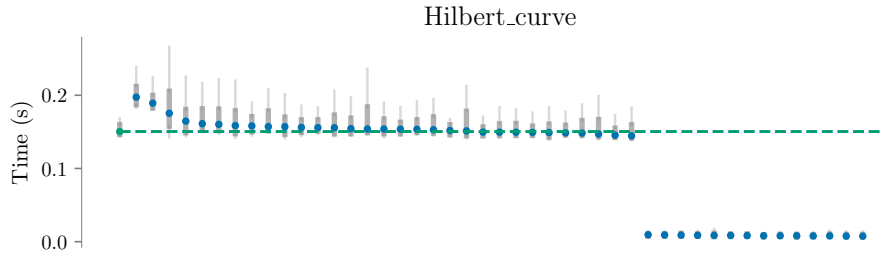


Figure 4.2: Execution time distributions for **Hilbert_curve** program and its variants. Baseline execution time mean is highlighted with the magenta horizontal line.

For the illustrated program, many diversified variants are optimizations (blue dots below the green bar). The plot is graphically clear, and the last third represents faster variants resulting from code transformations that optimize the original program. Our tool provides program variants in the whole spectrum of time executions, lower and faster variants than the original program. The developer is in charge of deciding between taking all variants or only the ones providing the same or less execution time for the sake of performance. Nevertheless, this result calls for using this timing spectrum phenomenon to provide binaries with unpredictable execution times by combining variants. The feasibility of this idea will be discussed in Section 4.3.

Answer to RQ2.

We empirically demonstrate that our approach generates program variants for which execution traces are different. We stress the importance of complementing static and dynamic studies of programs variants. For example, if two programs are statically different, that does not necessarily mean different runtime behavior. There is at least one generated variant for all executed programs that provides a different execution trace. We generate variants that exhibit a significant diversity of execution times. For example, for 169/239 (71%) of the diversified programs, at least one variant has an execution time distribution that is different compared to the execution time distribution of the original program. The result from this study encourages the composition of the variants to provide a more resilient execution.

4.3 RQ3. To what extent do the artificial variants exhibit different execution times on Edge-Cloud platforms?

Here we investigate the impact of the composition of program variants into multivariant binaries. To answer this research question, we create multivariant binaries from the program variants generated for Libsodium and QRcode corpora. Then, we deploy the multivariant binaries into the Edge and collect their execution times.

Timing side-channels.

We compare the execution time distributions for each program for the original and the multivariant binary. All distributions are measured on 100k executions of the program along all Edge platform nodes. We have observed that the distributions for multivariant binaries have a higher standard deviation of execution time. A statistical comparison between the execution time distributions confirms the significance of this difference (P-value = 0.05 with a Mann-Whitney U test). This hints

at the fact that the execution time for multivariant binaries is more unpredictable than the time to execute the original binary.

In Figure 4.3, each subplot represents the quantile-quantile plot [7] of the two distributions, original and multivariant binary. This kind of plots is used to compare the shapes of distributions, providing a graphical comparison of location, scale, and skewness for two distributions. The dashed line cutting the subplot represents the case in which the two distributions are equal, *i.e.*, for two equal distribution we would have all blue dots over the dashed line. These plots reveal that the execution times are different and are spread over a more extensive range of values than the original binary. The standard deviation of the execution time values evidences the latter, the original binaries have lower values while the multivariant binaries have higher values up to 100 times the original. Besides, this can be graphically appreciated in the plots when the blue dots cross the reference line from the bottom of the dashed line to the top. This is evidence that execution time is less predictable for multivariant binaries than original ones. This phenomenon is present because the choice of function variants is randomized at each function invocation, and the variants have different execution times due to the code transformations, *i.e.*, some variants execute more instructions than others.

Answer to RQ3.

The execution time distributions are significantly different between the original and the multivariant binary. Furthermore, no specific variant can be inferred from execution times gathered from the multivariant binary. Consequently, attacks relying on measuring precise execution times [?] of a function are made a lot harder to conduct as the distribution for the multivariant binary is different and even more spread than the original one.

Conclusions

This work proposes and evaluates an approach to generate WebAssembly program variants. Our approach introduces static and dynamic, variants for up to 11.78% of the programs in our three corpora, increasing the original count of programs by 4.15 times. We highlighted the importance of complementing static and dynamic studies for programs diversification. Moreover, combining function variants in multivariant binaries makes virtually impossible to predict which variant is executed for a given query. We empirically demonstrate the feasibility and the application of automatically generating WebAssembly program variants.

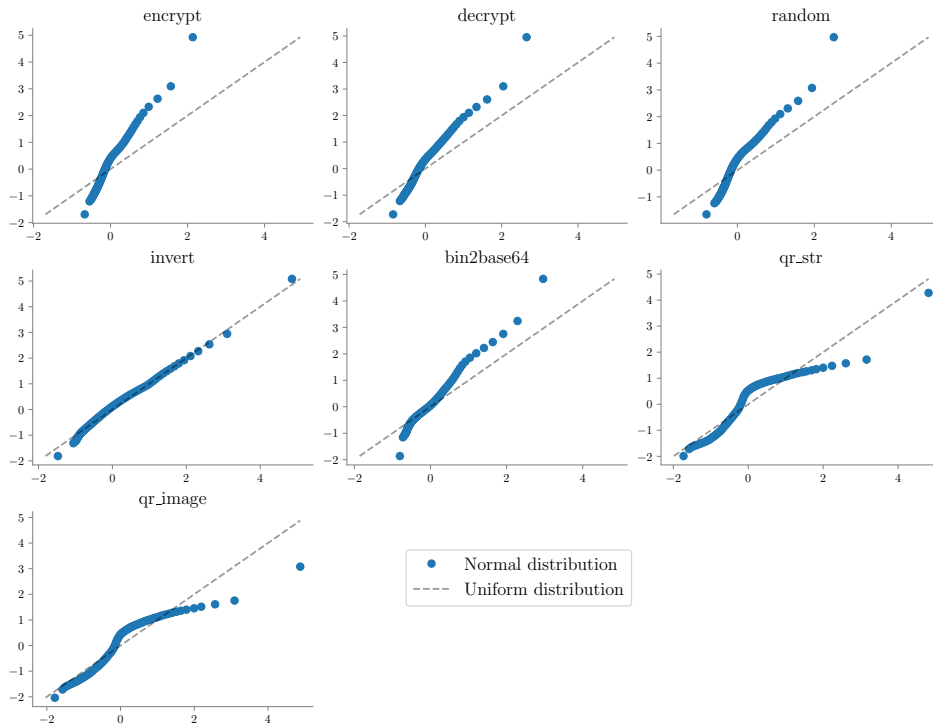


Figure 4.3: Execution time distributions. Each subplot represents the quantile-quantile plot of the two distributions, original and multivariant binary.

Bibliography

- [1] P. Runeson, E. Engström, and M.-A. Storey, *The Design Science Paradigm as a Frame for Empirical Software Engineering*. Cham: Springer International Publishing, 2020, pp. 127–147. [Online]. Available: https://doi.org/10.1007/978-3-030-32489-6_5
- [2] A. Hilbig, D. Lehmann, and M. Pradel, “An empirical study of real-world web-assembly binaries: Security, languages, use cases,” *Proceedings of the Web Conference 2021*, 2021.
- [3] J. Li, B. Zhao, and C. Zhang, “Fuzzing: a survey,” *Cybersecurity*, vol. 1, no. 1, pp. 1–13, dec 2018. [Online]. Available: <https://cybersecurity.springeropen.com/articles/10.1186/s42400-018-0002-y>
- [4] M. D. A. Maia, V. Sobreira, K. R. Paixão, R. A. D. Amo, and I. R. Silva, “Using a sequence alignment algorithm to identify specific and common code from execution traces,” in *Proceedings of the 4th International Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, 2008, pp. 6–10.
- [5] H. B. Mann and D. R. Whitney, “On a test of whether one of two random variables is stochastically larger than the other,” *Ann. Math. Statist.*, vol. 18, no. 1, pp. 50–60, 03 1947.
- [6] J. Cabrera Arteaga, P. Laperdrix, M. Monperrus, and B. Baudry, “Multi-Variant Execution at the Edge,” *arXiv e-prints*, p. arXiv:2108.08125, Aug. 2021.
- [7] R. Gnanadesikan and M. B. Wilk, “Probability plotting methods for the analysis of data,” *Biometrika*, vol. 55, no. 1, pp. 1–17, 1968.